Encapsulation in ES6:

Encapsulation is a fundamental concept in object-oriented programming that refers to the practice of hiding the internal details of an object and exposing only a public interface for interacting with it. In JavaScript, encapsulation can be achieved in ES6 (ECMAScript 6) through the use of classes, private properties and methods, and getters and setters.

One way to achieve encapsulation in JavaScript is to use the private keyword to define private properties and methods. However, this feature is not yet supported in ECMAScript 6 standard and is being proposed for future versions of the standard.

A common pattern for achieving encapsulation in JavaScript is to use an underscore prefix for properties and methods that should be considered private. This is a convention that indicates that the property or method is intended for internal use only and should not be accessed or modified directly by external code.

```
class Person {
  constructor(name, age) {
    this._personname = name;
    this._personage = age;
  }
  get personname() {
    return this._personname;
  }
  set personname(value) {
    this._personname = value;
  }
}

let john = new Person("John", 30);
console.log(john._personname); //undefined, it's a private property
console.log(john.personname); // Output: "John"
```

In this example, the properties _personname and _personage are considered private and are intended for internal use only. External code should not access or modify them directly. Instead, they can be accessed and modified through the name getter and setter.

Getters and setters can be used to control access to an object's properties and define custom behavior when those properties are accessed or modified; they provide a way to achieve encapsulation in JavaScript. By using getters and setters, you can hide the internal details of an object and expose only a public interface for interacting with it.

Inheritance in ES6:

Inheritance is a fundamental concept in object-oriented programming, and it is supported in JavaScript through the use of the class and extends keywords in ES6 (ECMAScript 6).

A class is a blueprint for creating objects, and it can have properties and methods that are shared by all instances of that class. The extends keyword is used to create a new class that inherits from an existing class.

Here is an example of inheritance in JavaScript using classes:

```
class Person {
  constructor(name, age) {
    this.personname = name;
    this.personage = age;
  }
  sayHello() {
    console.log(`Hello, my name is ${this.personname}`);
  }
}

class Student extends Person {
  constructor(name, age, major) {
    super(name, age);
    this.majorselected = major;
  }
  sayHello() {
    console.log(`Hello, my name is ${this.name} and I am studying
${this.majorselected}`);
  }
}

let john = new Student("John", 20, "Computer Science");
john.sayHello();
// Output: Hello, my name is John and I am studying Computer Science
```

In this example, the Student class extends the Person class, which means that it inherits all of the properties and methods of the Person class. The Student class also has its own constructor

and its own implementation of the sayHello method, which overrides the method inherited from the Person class.

The super keyword is used to call the constructor of the parent class, and it's mandatory to call it before using this keyword.

Inheritance is a powerful feature that allows you to create a hierarchical structure of classes, where common behavior and properties can be defined in a parent class and then specialized or overridden in child classes.

Polymorphism in ES6:

Polymorphism is a fundamental concept in object-oriented programming that refers to the ability of different objects to be treated as instances of the same class or interface. In JavaScript, polymorphism can be achieved in ES6 (ECMAScript 6) through the use of classes, inheritance, and interfaces.

One way to achieve polymorphism in JavaScript is through inheritance. When a class is defined as extending another class, it inherits all of the properties and methods of the parent class, and can be used in the same way as instances of the parent class.

```
class Shape {
    draw() {
        console.log("Drawing a Shape");
    }
}

class Circle extends Shape {
    draw() {
        console.log("Drawing a Circle");
    }
}

class Square extends Shape {
    draw() {
        console.log("Drawing a Square");
    }
}
```

```
let shape = new Shape();
let circle = new Circle();
let square = new Square();

let shapes = [shape, circle, square];

shapes.forEach(shape => shape.draw());
```

In this example, the classes Circle and Square extend the Shape class, and they can be treated as instances of the Shape class, because they inherit the draw method.

Another way to achieve polymorphism in JavaScript is through interfaces. An interface is a contract that defines a set of methods that must be implemented by any class that implements it. Classes that implement the same interface can be treated as instances of the same type and can be used interchangeably.

```
interface Drawable {
    draw();
}

class Circle implements Drawable {
    draw() {
        console.log("Drawing a Circle");
    }
}

class Square implements Drawable {
    draw() {
        console.log("Drawing a Square");
```